Distributed Leader Election Algorithms

by
Prince Francis

TH
CSE/1998/M
F859d

CSE
1998
M
FRA
DIS

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY KANPUR
FEBRUARY, 1998

Distributed Leader Election Algorithms

A Thesis Submitted

in Partial Fulfillment of the Requirements

for the Degree of

Master of Technology

by

Prince Francis

to the

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY, KANPUR

February, 1998

- 1 MAY 1998

RARY
UR

A 125387

CSE-1998-M-FRA-DIS

Entered in system

[illegible]
4-5-98

A125387

I.I.T. Kanpur.
Submitted on 10.2.98

CERTIFICATE

This is to certify that the work contained in the thesis entitled Distributed Leader Election Algorithms by Prince Francis has been carried out under my supervision and that this work has not been submitted elsewhere for a degree.

[illegible]

Dr. Sanjeev Saxena,

Associate Professor,

Department of Computer Science & Engineerin

Indian Institute of Technology, Kanpur.

Acknowledgments

I am deeply indebted to my thesis supervisor Dr. Sanjeev Saxena for his invaluable guidance and constant encouragement throughout this work. It has been a great pleasure working under him, who allowed me to choose a problem which I felt would be interesting. He has been an immense source of inspiration. I am grateful to him.

I am also thankful to my internal examiner Dr. Deepak Gupta for reading this thesis critically, pointing out some errors in writing and providing helpful comments. I also wish to thank him for giving a simpler description of algorithm for electing r leaders (see Section 2.4).

It would be very difficult to imagine the world without friends. Special thanks to Ratan, Anita, Murali, Biju, George, Ajith, David and all my other friends for giving a memorable company during my stay at IIT Kanpur.

Much of my work has been done in the cosy surroundings of IIT-K. It is simply superb. The trees, the peacocks, the birds have provided the needed enthusiasm and companionship while I was alone. Dear friends, I love you.

I do acknowledge the constant support and encouragement from my parents and loving sister. It was their support that gave me the spirit to carry on my study at IIT Kanpur.

Abstract

In this thesis we propose some new *distributed leader election algorithms* for synchronous *ring*, *complete*, *hypercube* and *cube connected cycles* networks. For the problem of electing r leaders, the generalized version of electing a single leader, we obtain an algorithm for synchronous ring with message complexity $O(n)$. For complete networks a series of algorithms, for message-time trade-off, is obtained; the algorithm uses $O(n^{1+\frac{1}{2^k}})$ messages and $O(k)$ time where k is an integer parameter to the algorithm. The algorithm is *optimal.* A randomized algorithm is proposed for complete networks that succeeds with 99% certainty in at the most five iterations with message complexity $O(n)$. An algorithm with time complexity $O(\log n)$ and message complexity $O(n)$ is proposed for hypercube networks. The algorithm is extended for electing r leaders at the expense of bit complexity, whereas the time and message complexities are the same as that of electing a single leader. The algorithm proposed for hy.percube networks is in ASCEND class and can be implemented on cube connected cycles.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The *Distributed Leader Election* problem is a very fundamental problem in distributed computations. In this thesis we propose some new distributed leader election algorithms for various network topologies and computation models. This chapter presents an overview of the distributed systems and distributed algorithm concepts, a survey of the problem under study, the significance of the problem and the organization of the thesis.

1.1 Distributed Algorithm Concepts and Terminology

1.1.1 Distributed Memory Systems

Distributed memory systems provide the motivation for the study of distributed algorithms. They comprise a collection of processors interconnected in some fashion by a network of communication links [5]. Depending on the system one is considering, such a network may consist of point to point connections, in which case each communication link handles the communication traffic between two processors exclusively, or it may comprise broadcast channels that accommodate the traffic among the processors in a large cluster. The processors do not share any memory, and the exchange of information among them can only be accomplished by message passing over the network of communication links.

The other relevant abstraction level in this overall panorama is the level of programs that run on the distributed-memory systems. One such program can be thought of as comprising a collection of sequential-code entities, each running on a processor, may be more than one sequential-code per processor. Such entities have been called tasks, processes or threads.

While at the processor level in a distributed-memory system there is no choice but to rely on message passing for communication, at the process level there are plenty of options. For example, processes that run on the same processor may communicate with each other through the explicit use of that processors memory or by means of message passing in a very natural way. Processes that run on different processors also have essentially these two possibilities. They may communicate by message passing by relying on the message-passing mechanisms that provide inter-processor communication, or they may employ those mechanisms to emulate the sharing of memory across processor boundaries. In addition, a myriad of hybrid approaches can be devised, including for example, the use of memory for communication by processes that run on the same processor and the use of message-passing among processes that do not.

The three classes of distributed-memory systems are long-haul computer networks, multiprocessors and networks of workstations.

1.1.2 Distributed Algorithms

A *distributed algorithm* executes as a collection of sequential processes, all executing their part of the algorithm independently, but coordinating their activity through communication [30]. Communication can take place through shared memory or by sending and receiving messages; in this thesis we assume that processes are communicating by message exchanges, and do not consider communication through shared memory. In parallel algorithms processes cooperate to achieve a single computational task, in order to complete the computation faster. Distributed algorithms are designed for tasks related to the physical dispersion of the processes and the data operated on and are meaningless within the context of a single sequential process.

Distributed algorithms are used to program distributed memory machines - long-haul computer networks, multiprocessors and networks of workstations. Other contexts in which these algorithms are used are operating systems and concurrent programming, reliable system design, programming languages, replicated databases and networks of automata. There is a tight connection between distributed algorithms research with research in graph theory and graph algorithms, because a distributed system is modeled by its underlying graph. Most of the problems studied in distributed algorithms are related to fault-tolerance, communication, synchronization and.control. A comprehensive survey of the origins of distributed algorithms and the typical problems studied has been done by Tel [30].

1.1.3 Processes

A *process* is an independent computing agent consisting of a *program* and facilities for *communication* with other processes or computing entities. The computations of a process are prescribed by its program, and depend only on the algorithm expressed by this program and the input/output events that take place. More precisely, the independence of a process refers to the fact that the next step in a computation by the process is determined solely by the algorithm and the history of the process. By the history of a process till at any point in time, we mean the computation that it has performed and the input/output events that have taken place in the process until this time. We can say that at each moment a process is in a certain *state*, which changes in each step of the computation performed by the process. The next step in a computation depends on the state of the process and its input/output events.

The local algorithm of a process may be *non-deterministic*. The processes in a distributed system are considered to be *sequential*. That is, the computations of a process proceed as totally ordered sequences of *steps*, or *events*. A process may consume input and produce output. We will only consider input and output in the form of *messages* from and to other processes in the system.

1.1.4 Messages and Channels

Messages are the only input a process receives from other parts of the system, and processes have no more information about what happens in the system other than what is implied by the messages that they receive. The state of other processes and the messages they exchange are unobservable, and information about it can only be collected from messages that are received at some (later) time.

The exchange of a message always takes place between two processes, the *sender* and the *receiver*. The sends *sends* by an output operation and the receiver *receives* by an input operation. The content of a message becomes available to the receiver when it receives the message. When the occurrence of the operations is timed on a global clock, a message is always sent before it is received. We distinguish between *synchronous* and *asynchronous* distributed systems. In synchronous distributed systems the operation of the entire system takes place in system-wide rounds. The processes are synchronized so that their steps take place simultaneously. In asynchronous systems no synchronization is present.

The exchange of a message between two processes requires that a connection between these two processes exists. A connection between two processes is called a *channel*, a *link*, or an *edge*. A channel always connects two processes, say p and q. A channel from p to q is a *unidirectional* (or *directed*) channel if p can use it to send message to q, but not vice versa. A channel between p and q is *bidirectional* (or *undirected*) channel if p can use it to send messages to q and also q can use it to send messages to p. We say that a network is unidirectional (directed) if its channels are unidirectional, and bidirectional if its channels are bidirectional. We say a channel is a *FIFO channel* or *obeys the FIFO rule*, if messages sent (in one direction) over the channel are always received in the same order as they are sent. A channel need not exist between any every pair of processes.

1.1.5 Network Topologies

It is convenient for the analysis of distributed systems to identify the network with the graph reflecting its connection structure. The nodes of the graph are the processes and the edges are the communication channels. The *degree* of a node is the number of edges incident to the node. For directed graphs the degree of a node is the sum of its *in-degree* (its number of incoming edges) and its *out-degree* (its number of outgoing edges). The neighbors of a node in a directed graph are its *in-neighbors* and *out-neighbors*. A *path* from p to q is a sequence $p = s_0, s_1, ..., s_k = q$ of nodes such that (for all $0 \leq i < k$) there is an edge from s_i to s_{i+1}; k is the length of the path. A (simple) *cycle* is a path from p to itself (in which no other node than p and no edge occurs more than once). The *distance* from p to q is the length of the shortest path from p to q. The *diameter* of the graph is the maximum distance from any node to any node.

We continue the discussion of graphs with the description of some particular graphs that are often used in the development of distributed algorithms. The number of nodes in the graph is denoted by n and it is assumed that $n > 1$.

▮ *Rings*

The n-node ring has nodes v_0 through v_{n-1} and edges v_iv_{i+1} for each i from 0 to $n-1$ (addition modulo n). The processes are thus connected in a ring-like topology. We say the ring is unidirectional if its edges are unidirectional and bidirectional if its edges are bidirectional. The importance of ring structure for distributed computing is three fold. First, because the number of edges of the ring is low, the ring structure allows for the development of algorithms with low communication cost. Second, because the ring structure is free of branching, a lot of potential non-determinism is eliminated, which allows for the development of simple algorithms. Third, a ring network in often used as a connection structure for local area networks, for example Token Rings. A ring structure can also be used as a control structure for distributed computations in arbitrary networks. It is then necessary to select a subset of the links that forms a Hamiltonian cycle in the network.

▮ *Trees*

A tree is a connected graph with $n-1$ edges. This number of bidirectional edges is the minimum required to connect n nodes, and as a result a tree is free of cycles. The importance of tree structure for distributed computing is twofold. First, because the number of edges of the tree is low, the tree structure allows for the development of algorithms with low communication cost. Second, while the branching structure allows a computation to proceed independently in parts of the network, the tree is free of cycles, so the branches of a computation never meet and conflicts are avoided. Thus the tree structure combines low communication cost with maximal independence of sub-computations. Like the ring, the tree is often used as a control (sub)-structure in arbitrary networks. It is then necessary to select a subset of the links that forms a spanning tree in the network.

■ *Stars*

A star is the unique tree with minimal diameter. A star has one central node, and every other node is connected only to this central node. The star network is a special case of a tree network, and thus has the advantages of tree networks. The star structure is used for local area networks that are organized as a central "server" node and several workstations. Computations in a complete network (see below) that are initiated by a single process often makes use of a subset of the edges that form a star with the initiating process as the center.

■ *Complete Networks*

A complete network is a graph in which every pair of nodes is directly connected by a bidirectional edge. It is the graph with minimum possible diameter (diameter is 1). The star network can be embedded in the complete network, and every node can serve as the central node of the star. Computations initiated by a single node can be restricted to a star with this node as the central node. A complete network has a large number of edges and each node has a large degree. Therefore the complete graph is never used as the physical structure of large processor networks. But, it is of very high theoretical importance and it has studied much in the literature. They provide bounds for more practical networks.

▪ *Grid and Torus Networks*

The $n \times m$ grid network contains n times m processors, arranged in a rectangle, and the links between neighboring processes in horizontal and vertical directions. The $n \times m$ grid with wrap-around, or $n \times m$ torus, has additional connections between processes on the right and the left border of the rectangle, and between processes on the upper and lower border. Grid and torus networks are popular network structures for multiprocessor computers. Matrix computations are well suited for this structure. The two dimensional structure can be generalized to higher order grids and toruses.

▪ *Hypercube Networks*

In a hypercube network the nodes correspond to the corners of an n-dimensional cube and the links are the edges of the cube. For a precise description of the hypercube with $n = 2^m$ nodes, nodes can be identified with bit strings of length m. An edge exists between two nodes if the corresponding bit strings differ in exactly one position. Again this structure is popular for multiprocessor computers. *Cube connected cycles* is a variation of hypercube structure. In the cube connected cycles network each corner of the cube is replaced by a ring of m nodes, and each of these has one of the links of the cube network attached to it.

▪ *General Networks*

The term *general networks* refers to all connected graphs. An algorithm for general networks runs on every arbitrary connected network.

1.2 Complexity

The execution of an algorithm consumes the resources of the network. The complexity analysis of an algorithm aims at determining how much of a certain resource is used by the algorithm.

1.2.1 Worst Case and Average Case Complexity

Different executions of an algorithm may use different quantities of resources. By *worst case complexity* of an algorithm we denote the maximum resource consumption, over all possible executions of the algorithm. By the *average case* complexity we mean the average over all possible executions. In this thesis we use only worst case complexities for deterministic algorithms.

1.2.2 Complexity Measures

The *communication complexity* of an algorithm can be measured in two ways: by the number of messages and by the number of bits exchanged. The message complexity is the number of messages sent in an execution of the algorithm. The *bit complexity* of an algorithm is the total number of bits included in all messages in the execution together. Because the size of the messages may vary among different algorithms the message complexity is not immediately related to the bit complexity. Expressions representing orders of magnitude are used for message complexity as well.

The *time complexity* of an algorithm is the amount of time an execution takes. In synchronous systems this is counted as the number of rounds over which the actions of the execution are spread. The time complexity of an asynchronous algorithm is usually computed under the assumptions that local computations in a process costs time 0, and message delay is 1 for every message.

Communication and time complexity are the most important measures we use. The *storage complexity* is the amount of local memory that a process must have to run its part of the distributed algorithm. We always ignore the cost of local computations within a process.

1.2.3 Complexity Parameters

The complexity of an algorithm is usually expressed in terms of parameters that describe the size of the problem. For problems concerning networks the important parameters are: n (the number of processes), e (the number of edges) and D (the diameter of the network).

Complexities are often expressed only in order of magnitude. Let f and g be functions from N to N, where N is the set of natural numbers. We say that $f(n)$ is $O(g(n))$ if there exists a real constant $c > 0$ such that for all but finitely many $n, f(n) \leq c.g(n)$. We say that $f(n)$ is $\Omega(g(n))$ if there exists a real constant $c > 0$ such that for all but finitely many $n. f(n) \geq c.g(n)$. We say that $f(n)$ is $\Theta(g(n))$ if $f(n)$ is both $O(g(n))$ and $\Omega(g(n))$.

1.3 Leader Election Problem and Literature Survey

A difficult problem area in distributed systems is how to control the execution of a distributed system. Processes, by their nature, act independently of the other processes. It is, however, required that they cooperate in a consistent way. In some distributed algorithms there is a need for exactly one special "leader" process, which is to execute an algorithm different from those in other processes. If there is no pre-determined leader process, an election procedure must be executed to choose one of the processes to act as this leader. The problem was first posed explicitly by LeLann [17]. The problem is usually considered under the assumption that each process has a unique identity, and a total order on the identities is available. The problem is also known as the Leader Finding or Extrema Finding problem. In the later variant it is required that the process with the largest process identity is chosen as the leader.

Let the network consists of n nodes. A leader is one of these n nodes that all other nodes acknowledge as being distinguished to perform some special task. The *leader election problem* is the problem of choosing a leader from a set of *candidates*. We assume that the set of all identifications is totally ordered by relation "<". This assumption is fundamental in the approaches to leader election that take the leader to be the candidate with greatest identification. However, even if this is not the criterion, the ability to compare two candidates' identifications is essential to break ties that may occur with the criterion at hand. In fact, this is really why unique identifications are needed in the first place. In their absence, any criterion to select a leader from the set of candidates might deadlock for the absence of a tie breaker.

The importance of electing a leader in a distributed environment stems essentially from the occurrence of situations in which some centralized coordination must take place, either because a technique to solve the particular problem at hand in a completely distributed fashion is not available, or because the centralized approach offers more attractive performance. Problems for which satisfactory techniques of a completely distributed nature are not available include the many recovery steps that have to be taken after the network undergoes a failure (or a topological change, in broader terms). A leader in this case is needed to coordinate, for example, the re-establishment of allocation and routing functions. Examples to illustrate the importance of electing a leader when the centralized approach to a particular problem proves more efficient than the distributed one come from the area of graph algorithms, for example establishing a minimum spanning tree.

The problem arises in many other situations. Sometimes a distributed computation must be carried out under the supervision of exactly one process, for example - routing table distribution. Some other algorithms may rely on the circulation of one or more special messages, called tokens, through the network. If a token is lost, a new token must be generated by exactly one process; LeLann [17] discusses the utility of the problem in the generation of a token in a ring of processes.

Motivated by LeLann's work, the problem was first studied for the special case where the network topology is a ring. LeLann's algorithm used $O(n^2)$ messages for an election on a ring with n processes. The algorithm by Chang and Roberts [8] used the same number of messages in the worst case, but only $O(n \log n)$ messages on the average. Peterson [21] showed that there exists an algorithm which uses only $O(n \log n)$ messages in the worst case. Frederickson and Lynch [9] have given an $O(n)$ message algorithm for synchronous ring network in $O(n)$ time, under the assumption that processor IDs are chosen from some countable set. A bit optimal election algorithm in synchronous rings with unknown size was proposed by Bodlaender and Tel [6]. The bit complexity is linear in the number of processors. The time complexity was polynomial in the number of processes, but exponential in the smallest identity of any process. Abrahamson *et al.* [1] have constructed a randomized algorithm for asynchronous rings. If processors have unique identities with a maximum of m bits then the expected number of communication bits sufficient to elect a leader with probability 1, on a ring of (unknown) size n is $O(nm)$. Election on chordal rings was studied by Kalamboukis and Mantzaris [16]. They presented an election algorithm for a class of chordal rings with constant number of chords at each processor and $O(n \log \log n)$ message complexity. Marchetti-Spaccamela [19] proposed a new protocol schema that can be specialized to obtain several protocols with different communication-time characteristics when the network is ring shaped and communication between processors is synchronous.

Gallager, Humblet, and Spira [12] have shown that in a network of arbitrary, unknown topology consisting of n nodes and e channels, a leader can be elected by exchanging $O(n \log n + e)$ messages in the worst case. Korach *et al.* [15] have constructed more efficient algorithms with a suitable topology: $O(n \log n)$ messages suffice for completely connected networks, rings, and complete bipartite networks.

Leader election in complete networks has been studied extensively with different models and assumptions. Afek and Gafni [2] give $O(n \log n)$ messages synchronous and asynchronous algorithms. The time complexity of the synchronous algorithm is $O(\log n)$ while that of asynchronous algorithm is $O(n)$; the algorithms are comparison based. Singh [27] proposed a leader election protocol in asynchronous complete networks with a sense of direction. A network has a *sense of direction* if there exists a directed Hamiltonian cycle and each edge incident at a node is labeled with the distance of the node at the other end of this edge along this Hamiltonian cycle. The proposed algorithm requires $O(n)$ messages and $O(\log n)$ time. Singh [25] also describes a protocol that requires $O(n \log n)$ messages and $O(\frac{n}{\log n})$ time for asynchronous complete networks. Leader election in the presence of link failures has been studied by Singh [26]. He gave a message optimal algorithm with $O(n^2)$ messages under the assumption that up to $\frac{n}{2} - 1$ links incident to each node each node may fail during the execution of the algorithm. Message transmission times is assumed to be finite but unpredictable. Moreover, the channels may lose messages. The problem in asynchronous complete networks with faulty links has also been studied by Sayeed and Abu-Amara [29]. They assumed *Byzantine* failures, that is, channels fail by altering messages, sending false information, forging messages, losing messages at will, and so on. The processors were assumed to be reliable. The algorithm can tolerate up to $\lfloor \frac{n-2}{2} \rfloor$ faulty channels.

Brunekreef *et al.* [7] studied the election problem in broadcast networks. Singh [28] and Afek and Stupp [4] studied the problem of real-time leader election in a shared memory environment. Elections in anonymous networks was studied by Afek and Mathias [3]. In anonymous networks processors do not have an identity. Consequently, on anonymous networks randomization techniques must be used.

1.4 Organization of the Thesis

The rest of the thesis is organized as follows. In chapter 2 we present an efficient algorithm for electing r leaders, a generalized version of single leader election, in synchronous ring networks. In Chapter 3 we propose an optimal algorithm for complete networks. We also present an algorithm which runs in $O(\log\log n)$ time. A randomized algorithm for complete networks appears in Chapter 4. In Chapter 5 algorithms for hypercube networks are presented. We end the thesis with concluding remarks which include some suggestions for further work.

Chapter 2

An Efficient Algorithm for Electing r Leaders in a Synchronous Ring

2.1 Introduction

In this chapter we give an efficient algorithm for electing r leaders out of n processors ($r \leq n$) in the case of a synchronous ring network[1]. The leaders have to be elected from a subset of n processors known as the set of *competing processors.* Ring networks have been studied extensively in the literature. Fredrickson and Lynch [9] give an a $O(n)$ message algorithm for electing a single leader. We generalize the problem of electing a single leader to electing r leaders after [22]. There are many situations where more than one leader has to be elected in a network. Our algorithm requires only $O(n)$ messages (same as that of a single leader election problem) and a total bit complexity

[1]Some results in this chapter appeared in - P. Francis and S. Saxena, "Efficient Algorithm for Electing r Leaders in a Synchronous Ring", *Proceedings of the Seventh National Seminar on Theoretical Computer Science, June 1997.*

of $O(nr\log m)$ where the IDs are taken from $\{0, 1, ...m-1\}$.

We assume that the communication network consists of n nodes that are connected as n-node ring. The *n-node ring* has nodes v_0 through v_{n-1} and edges v_iv_{i+1} for each i from 0 to $n-1$ (addition modulo n) [30]. The processors are thus connected in a ring-like topology. We say the ring is unidirectional if its edges are unidirectional, and bidirectional if its edges are bidirectional. We assume the ring is *unidirectional.* We assume *synchronous* model of communication, ie., a global clock is connected to all nodes and computation takes place through steps. Each step of the computation is known as a *round.* We can assume that the communication is counter-clockwise. The n-node ring consists of n nodes whose IDs are *unique* and taken from a totally ordered set $\{0, 1, 2, ..., m-1\}$, $m \geq n$. We shall refer to the node with $ID\ i$ as "node i". We use the terms *node* and *processor* synonymously.

2.2 The Algorithm

In this section we present the algorithm. The algorithm starts when x out of n processors wake up to compete in the election process, where $x \geq r$, r being the number of leaders to be elected. Each processor competing in the election process spawns a *message token* having $\log m + r\log m + 2$ bits. First $\log m$ bits represents the ID of the processor which generated the token, called the *owner* of the token. Next $r\log m$ bits are used as a *current leader list*; the list has sufficient space to store IDs for the current r leaders during the execution of the algorithm. Each leader ID is encoded in $\log m$ bits. The third field consists of a single bit called *zero flag* to distinguish between empty $\log m$ bits and the encoded ID for processor 0 in the second field. Whenever the ID 0 is entered or deleted in the current leader list, the zero flag is set or reset respectively. The zero flag bit is also used by other processors while recognizing the current leaders in the list.

The fourth field also consists of a single bit, called *delay flag*. We will see its use in the algorithm.

When a processor i spawns a token, it sets the owner field to i, enters its ID i in the empty current leader list and sets the delay flag. If the processor ID is 0, then it sets the zero flag also. The spawned token is sent to the next processor in the ring (counter-clockwise direction). No processor is allowed to enter the set of competing processors (x out of n) after it has received a message from an awakened processor. As mentioned earlier, the ring is assumed to be *synchronous* and each round takes one unit of time. Each token travels with different speed across the ring. The token, whose owner is i is delayed 2^{n-i-1} rounds at each processor j,$(0 \leq j \leq n-1)$, before j sends it to the next processor. If another token overtakes the first token by this time, then the first token will not be sent to the next processor. In this case we say that the token is purged or killed by the processor j. Thus any slower token overtaken by a faster token is killed. The algorithm executed by a processor i which receives a token, whose owner is j, is given in Figure 1.

Whenever a competing processor i gets a token whose owner is j, it checks whether its (i's) ID is there in the current leader list. If its ID is not in the list and if $i > j$ then kill the token. That is, a token is killed by a processor whose ID is greater than the owner of the token. If the owner of the token is greater than the ID of the processor then the processor will try to add its ID to the current leader list if possible. Processor i can add its ID only when the number of entries is less than r or there exists an ID k in the list such that $k < i$ (by replacing k with i). As we mentioned, token i is delayed by the processor j by 2^{n-j-1} rounds. This technique reduces the overall message complexity of the algorithm, since the token is sent again only if it is not overtaken by another faster token during the delay of 2^{n-j-1} rounds. If processor ID is there in the list, then the token has completed a trip across the ring. Therefore the processor declares itself as

```
if the current leader list is empty then purge the token
                      and terminate the process;
if ($i$ is not in the current leader list) then
   if ($i > j$) then purge the token;
   else/* $i \leq j$ */
   if ($i < j$) and (delay flag is set) then
      if (number of leaders in the current leader list < $r$) then
         add $i$ to the current leader list;
      elseif (min(leader list) < $i$) then
          replace min(leader list) by $i$;
   if delay flag is set then
            delay the token for $2^{n-j-1}$ rounds;
   if no token has overtaken then
             pass the token;
else/* if $i$ is in the current list */
    declare itself as one of the leaders;
    remove $i$ from the current leader list;
    reset the delay flag;
    pass the token;
```

Figure 1: Algorithm executed by a node i on receiving token owned by j

one of the leaders. It removes its ID from the list, resets the delay flag and passes the token. Once the list contains the actual leaders to be elected, then there is no need for any delay, since by that time all other tokens except the current one must have been killed. When all actual leaders come to know that they are elected then this final token will have empty list, since each processor removes its ID after knowing its success. The token which consists of empty list is killed by the next processor, thus terminating the algorithm. A processor not competing for a leader simply passes the tokens to the next processor.

2.3 Complexity Analysis

In the worst case all n processors wake up simultaneously and spawn a token each. Clearly, the final token accounts for $O(n)$ messages, since it has to go through $O(n)$ nodes. By the time this token completes its travel another token j could travel at most $n/2^{n-j-1}$ distance. The number of messages will be the sum of all such messages for each j. Summing the contribution of each token, the worst case message complexity is $O(n)$. The corresponding bit complexity is $O(nr \log m)$, since each message(token) consists of $O(r \log m)$ bits. The overall computing time of the algorithm is $O(2^{n-x-1}.n)$, since $O(2^{n-x-1}.n)$ rounds are required for the final token to be killed, where x is the largest ID among the competitors.

2.4 Electing r Leaders in a General Graph

The problem of electing a single leader has been studied extensively in the literature. In the previous section we discussed the more general problem of electing r leaders out of n nodes in a network, $r \leq n$. The generalization of the problem has its significance. In general, the problem of electing r leaders and the problem of electing a single leader are not equivalent (in terms of complexity) problems. It can be seen that for some topology, for example, for a complete network, it may happen that these two problems are equivalent in terms of complexity. In a complete network r leaders can be elected in the same complexity bound as that of electing a single leader. From Chapter 3, the lower bounds for complete networks is $n^{1+\epsilon}$ messages in constant time. Suppose we have elected a single leader in the network. Now, in the next round, all other nodes send their IDs to the leader node. From the list of IDs of all nodes in the network, the leader can select the remaining $r-1$ leaders and inform the $r-1$ leaders that they are elected as leaders in the network. This can be done at the expense of $O(n)$ messages in constant time. Thus in a complete network it so happens that these two problems are equivalent in terms of complexity bounds.

Consider the ring network (unidirectional). The lower bound for electing a single leader is $O(n)$ messages. If we adopt the strategy for electing r leaders in the complete network, we first elect a single leader using $O(n)$ messages. Now all nodes have to send their IDs to the elected leader. In a ring network, this will take $1 + 2 + 3 + ...(n-1) = O(n^2)$ messages,assuming that each message consist of $O(\log n)$ bits. Similarly, $O(n^2)$ messages will be taken by the leader to inform the r elected leaders. We can not do this using $O(n)$ messages to match the lower bound of electing a single leader. In our algorithm for electing r leaders in this chapter, we achieve the same bounds for both messages and time by sacrificing the bit complexity (due to increased length of messages). The bit complexity of our algorithm is $O(nr \log m)$, which is not independent of r.

A simpler description of the algorithm for electing r leaders on ring is [see acknowledgments]: Each message is of $O(r \log n)$ bits. First elect a single leader then by passing a token and each candidate processor which appending its ID to it. This gives an esthetically pleasing algorithm for electing r leaders.

2.5 Summary

In this chapter we have proposed a distributed algorithm to elect r leaders out of n nodes in a synchronous ring network; the algorithm takes $O(n)$ messages and has a bit complexity of $O(nr \log m)$ bits. It has been shown that, in general, the problems of electing a single leader and that of electing r leaders are not equivalent. In general, the later problem is computationally more complex.

Chapter 3

An Optimal Algorithm for Complete Networks

3.1 Introduction

Leader election problem has been studied extensively in different models and networks [11, 9, 22, 23, 24, 10]. There are many problems such as spanning tree construction and computing a global function, which are equivalent to leader election in terms of message and time complexities. In this chapter, we study the leader election problem in complete networks. Although complete networks are not practical, they have been studied extensively as they provide bounds for more practical networks. In this chapter, we propose two algorithms. First we give an *optimal* algorithm for electing a leader in a complete network. It combines the best features of two existing algorithms, the algorithm which requires $O(n^2)$ messages and constant time [25] and the algorithm which requires $O(n \log n)$ messages and $O(\log n)$ time [2]. Our general algorithm requires $O(k)$ time and uses $O(k.n^{1+\frac{1}{2^k}})$ messages, where k is a constant; by fixing a value of k, in particular, our algorithm can elect a leader in $O(1)$ time using $O(n^{1+\epsilon})$ messages. This is optimal and meets the lower bound results of [2]. We propose a second algorithm which requires $O(\log\log n)$ time and *linear* number of messages, by making the assumption that each node knows the IDs of its neighboring nodes.

In a *complete network*, each pair of nodes is connected by a bidirectional communication link. If the network consists of n nodes, every node is connected by $n-1$ bidirectional communication links to all the other nodes. Nodes have unique IDs taken from the set $\{0, 1, 2, \ldots, m-1\}$, but no node knows the ID of any of its neighbor. However, every node knows the value n before the algorithm starts. We assume *synchronous* mode of communication. In this model a global clock is connected to all the nodes of the network. The time interval between two consecutive pulses of the clock is a round. At the beginning of each round, each node decides, according to its state, what messages to send and on which links to send. Each node then receives any message sent to it in this round and uses all the received messages and its state to decide its next state.

The nodes of the network are initially asleep. An arbitrary subset of nodes, called *competitors*, wakes up spontaneously and starts the algorithm by sending messages over the network. When the message exchange terminates, a leader is distinguished from all other nodes. It is also assumed that the processing time of messages is negligible compared to the communication time.

The *message complexity (communication cost)* of the algorithm is the worst-case total number of messages sent during the execution of the algorithm, where a message contains at most $O(\log m)$ bits. The *time complexity* is the worst-case total number of time units from the first to the last message transmission due to the algorithm, where a time unit is one round.

```
Candidate program
          $untraversed$ <- $E$ ;
          $level$ <- $-1$;
          In Each $round$ do
                  $level$ <- $level + 1$;
                  if $level$ is even
                  then
                       $l$ <- $\frac{2^{level/2}}{2^k}$;
                       if $l < 1$ then
                          Send ($level, id$) over $p = n^l$ links from $untraversed$
                       else if $l \geq 1$ then
                          Send ($level, id$) over, $p = n$ links (all)
                  else
                       Receive all acknowledgment type messages
                       if received less than $p$ acknowledgments then Stop
                       else if $level = 2k + 1$ then Elected, Stop.
          End each $round$.
```

Figure 1: Candidate Program

3.2 The Algorithm

In this section, we give the distributed algorithm for election in synchronous complete networks. Our algorithm not only combines the best features of the trivial $O(1)$ time with $O(n^2)$ messages algorithm and the $O(\log n)$ time with $O(n \log n)$ messages algorithm given in [2] but will be shown to be worst-case optimal. The algorithm is initiated by any subset of nodes, each of which is a *candidate* for leadership. Each candidate tries to capture all other nodes by sending messages on all the links incident to it. The algorithm terminates when a candidate has succeeded in capturing all its neighbors. This candidate becomes the leader. To guarantee that only one node is elected, all candidates but one are killed.

All candidates use a variable *level* to estimate the number of nodes they have already

```
Ordinary program
        $l^*$ <- $nil$; /* the link over which the lexicographically largest $(level, id)$ a
        $level$ <- $-1$;
        $owner\_id$ <- $id$; /* its own id */
        In Each $round$ do:
                Send an acknowledgments over $l^*$;
                $level$ <- $level + 1$;
                Receive candidate messages $(level, id)$ over all links $l$;
                Let $(level^*, id^*)$ be the lexicographically largest
                $(level, id)$ candidate and $l^*$ the link over which it arrived;
                if $(level^*, id^*) > (level, owner\_id)$
                then
                    $(level, owner\_id)$ <- $(level^*, id^*)$;
                else
                    $l^*$ <- $nil$;
        End each $round$.
```

Figure 2: Ordinary Program

captured. The variable *level* is used by candidates to "duel" each other. For captured nodes variable *level* keeps track of the highest *level* of candidates they have observed. That candidate is their *owner*. All variables *level* are initialized to 0. A candidate that sees a node with a larger *level* than its own is eliminated from candidacy. However, if the candidate's *level* is larger or equal, the node's *level* is replaced by the candidate's *level*. The candidate may then claim the node and try to eliminate the previous owner of the node.

The *level* of a candidate is the number of *rounds* since it started the algorithm. To simplify the algorithm, every initiator node spawns two processes, the *candidate process* and *ordinary process*. The two processes are connected to each other by a bidirectional logical link which behaves like a physical link. A node awakened by receiving a message from the algorithm spawns only an ordinary process. Candidate processes communicate only with ordinary processes and vice versa. Thus the communication topology can be viewed as a complete bipartite graph, with the candidate processes on one side and n ordinary processes on the other. Henceforth, the term *candidate* will be used interchangeably for both the process and its initiating node. Messages received from candidate processes are forwarded to the ordinary process. Messages received from ordinary processes are forwarded to the candidate process.

The algorithms executed by candidate and ordinary programs are given in Figure 1 and Figure 2 respectively. Every live candidate at level $2i$, $i \geq 0$, tries to capture $n^{(2^i/2^k)}$ new ordinary processes by sending them messages containing its *level* and id, where k is an integer parameter to the algorithm. If in level $2i+1$ the candidate receives acknowledgments from all the ordinary processes it tries to capture, then it proceeds as a candidate to the next level. Otherwise the process is eliminated from candidacy. The level of a candidate in incremented after every round. Every ordinary process has *level* and *owner-id* variables that contain the level and id of the highest-level candidate the process from which it has received a message from (*level* ties are resolved by selecting the highest id). In every round, every ordinary process first increases its level by one, to reflect the owner's actual level, and inspects the newly received messages to update its level and owner-id if necessary. If an update occurs, the ordinary process acknowledges its new owner. In the algorithm, E is the set of edges incident to a candidate process. Every candidate maintains a list of edges, called *untraversed*, which it has not yet traversed in any direction.

Our algorithm runs with a parameter k; k being a positive integer. Depending on the different values of k we get a series of algorithms with varying time-message tradeoffs. The general algorithm runs in $O(k)$ time with $O(kn^{(1+(1/2^k))})$ messages.

3.3 Analysis

We observe the following facts.

Fact 3.1 *The level of every ordinary node strictly increases from one round to the next.*

This follows immediately from the algorithm for ordinary node processes.

Fact 3.2 *At most $n^{(1-(\sum_{x=0}^{i-1} 2^x/2^k))}$ candidates reach level $2i$, $1 \leq i \leq k$.*

This follows from Fact 1 and the observation that every ordinary node acknowledges at most one message in which the level is i, the set of $n^{\frac{2^i}{2^k}}$ nodes that are captured by each candidate that has reached level $2i$ are disjoint.

Fact 3.3 $2k+1$ *rounds after the algorithm has started, a unique candidate captures all the nodes and is elected as the network leader.*

This holds because all the messages of the final node get acknowledged, and once a node has acknowledged, it will not acknowledge any other message. The result now follows from the algorithm.

The time complexity of the algorithm is clearly $O(k)$, since algorithm requires $2k+1$ rounds to finish the algorithm. Since every node sends at most one acknowledgment to a candidate in level $2i$, the total number of acknowledgments is kn, each of $O(1)$ bits. From Fact 2, at most $n^{(1-(\sum_{x=0}^{i-1} 2^x/2^k))}$ candidates reach level $2i$. At level $2i$ a surviving candidate tries to capture $n^{(2^{level/2}/2^k)} = n^{2^i/2^k}$ nodes by sending messages. Total messages in every even round is given by,

$$
\begin{aligned}
msgs &= n^{(1-(\sum_{x=0}^{i-1} 2^x/2^k))} \times n^{(2^i/2^k)} \\
&= n^{(1+(2^i-\sum_{x=0}^{i-1} 2^x/2^k))} \\
&= n^{(1+(1/2^k))}
\end{aligned}
$$

Since there are $2k+1$ rounds, the message complexity is $O(kn^{(1+(1/2^k))})$ and time complexity is $O(k)$. By fixing a value of k, we get a constant time algorithm which uses $O(n^{1+\epsilon})$ messages. Our algorithm is optimal. The lower bound proof for the number of messages and the time required for complete networks is given in [2]. The lower bound theorem is,

Theorem 3.1 *[2] Any stopping-execution of an election algorithm in a synchronous complete network of n nodes which terminates in less than $\frac{1}{2}.\log_c n$ rounds, contains at least $\frac{c-1}{2.\log c}.n\log n$ events.*

The "events" is same as the messages required and the "rounds" same as the time required in the synchronous model. Our algorithm runs in $O(k)$ time. Thus $k = \frac{d}{2}\log_c n$ or $c = n^{\frac{d}{2k}}$ Now the optimal algorithm should have at most $\frac{c}{\log c}.n\log n$ messages. Substituting for c, the optimal algorithm should have at most $\frac{n^{\frac{d}{2k}}}{\frac{d}{2k}\log n}.n.\log n = \frac{2k}{d}n^{1+\frac{d}{2k}}$ messages. This shows that our algorithm is optimal.

3.4 $O(\log\log n)$ Time Algorithm

In this section we propose a $O(\log\log n)$ time algorithm for electing a leader in a complete network, under the assumption that each node knows the *ID*s of its neighbors. The algorithm works with *linear* number of messages. The algorithm is based on *recursive division* of the network into sub-networks.

The algorithm has a parameter s. We divide the network into $\frac{n}{s}$ sub-networks each of size s. A node with ID x is put in $(\frac{x}{s})^{th}$ set of the sub-networks. This division is represented as (n,s). A leader is elected in each sub-network recursively. At the end of recursion we use the $O(1)$ time and $O(n^2)$ messages algorithm described in section 2 of chapter 4. Now each sub-network consists of $n' = \frac{n}{s}$ nodes. Now, we again divide the sub-networks consisting of $n' = \frac{n}{s}$ nodes into sub-sub-networks where each sub-sub-network consist of $s' = \frac{s^2}{2}$ nodes. This recursive division is represented as,

$$(n, s) \rightarrow (\tfrac{n}{s}, \tfrac{s^2}{2}) \rightarrow (\tfrac{2n}{s^3}, \tfrac{s^4}{8}) \rightarrow \ldots\ldots.$$

the recursive division is continued until $n' = s'$. When $n' = s'$ we stop the recursion and uses the trivial $O(1)$ time and $O(n^2)$ message algorithm to elect a leader.

To analyze the algorithm let us define the function H_{k+1} as follows.

$$\begin{aligned} H_{k+1} &= 2H_k + k, \text{ for } k \geq 3 \\ &= 1, \text{ for } k = 2 \\ &= 0, \text{ for } k = 1 \end{aligned}$$

Solution of this recurrence is given by,

$$H_k = 2^k - k - 1 \tag{3.1}$$

In the k^{th} recursive step the number of nodes left in a sub-group is given by,

$$n' = \frac{n2^{H_k}}{s^{2^k-1}} \tag{3.2}$$

Similarly, the unit of division s' is given by,

$$s' = \frac{s^{2^k}}{2^{H_k+k}} \tag{3.3}$$

We stop the recursive division as soon as $n' = s'$.

$$\begin{aligned} \frac{n2^{H_k}}{s^{2^k-1}} &= \frac{s^{2^k}}{2^{H_k+k}} \\ n2^{2H_k+k} &= s^{2^{k+1}-1} \\ n2^{H_{k+1}} &= s^{2^{k+1}-1} \end{aligned}$$

$$\log n + H_{k+1} = (2^{k+1} - 1)\log s$$

$$\begin{aligned}
\log n + 2^{k+1} - k - 1 &= (2^{k+1} - 1)\log s \\
&= 2^{k+1}\log s - \log s \\
\log n &= (k + 1 - \log s) + 2^{k+1}(\log s - 1) \\
&\geq 2^{k+1}(\log s - 1)
\end{aligned}$$

$$k = \log\log n - \log(\log(s) - 1) - 1 \tag{3.4}$$

We see that k is $O(\log\log n)$ which is the number of rounds required for the algorithm. A sub-network consisting of s nodes requires s^2 messages to elect a leader. Since there are $\frac{n}{s}$ such sub-networks, total messages required is ns, which is *linear*. From equation (3.4) we see that $s \geq 4$. This algorithm beats the lower bound of [2].

3.5 Summary

We have obtained a message-optimal time-message tradeoffs in synchronous model; in particular we can get an optimal constant time algorithm using $O(n^{1+\epsilon})$ messages. We have also obtained a $O(\log\log n)$ time algorithm with linear number of messages by taking the assumption that each node knows the IDs of its neighbors.

Chapter 4

A Randomized Algorithm for Complete Networks

4.1 Introduction

In this chapter we give a randomized algorithm for distributed leader election problem in a complete network. A *randomized algorithm* is is allowed to makes random choices during execution [18]. There are two principal advantages of randomized algorithms. The first is performance - for many problems, randomized algorithms run faster than the best known deterministic algorithms. Second, many randomized algorithms are simpler to describe and implement than deterministic algorithms of comparable performance. We distinguish between two classes of randomized algorithms: *Las Vegas* and *Monte Carlo* [14]. A randomized algorithm of the Las Vegas type will always generate the correct answer. A randomized algorithm of the Monte Carlo type is allowed to make errors, but only with a "small" probability. *Randomized Quick-sort* is a well known example for a randomized algorithm. Consider sorting a set S of n numbers into ascending order. We pickup a member y of S and use the following scheme. We partition $S - \{y\}$ into two sets S_1 and S_2, where S_1 consists of those elements of S that are smaller than y, and S_2 has the remaining elements. We recursively sort S_1 and S_2, then output the elements of S_1 in ascending order, followed by y, and the elements of S_2 in ascending order. It can be shown that the expected time complexity of this algorithm is $O(n \log n)$.

Our randomized algorithm for electing a leader in a complete network is simple and efficient. A randomized algorithm may fail during its execution. It may be re-executed until we succeed. Our algorithm succeeds with 0.99 probability of success (with 99% certainty) in at the most five iterations with linear number of messages. The IDs are distinct and are taken from the set $\{0, 1, ..., m-1\}$, $m \geq n$, where n is the number of nodes in the network. A node knows its neighbors but not their IDs. We assume *synchronous* mode of communication and that each node knows the approximate number of the candidates.

4.2 The Algorithm

First we give the trivial constant time algorithm with $O(n^2)$ message complexity [25]. In this algorithm, a *candidate* attempts to capture all other nodes. The node that is able to capture all other nodes declares itself the leader. On waking up, a candidate sends its identity in an *elect* message on all incident edges. When a node j receives an *elect(i)* message over edge e, it behaves as follows.

If j is a candidate and $j > i$, then no response is sent over e.

Otherwise, j sends an *accept* message over e.

A node that receives an *accept* message on all incident edges declares itself the leader and notifies all nodes of this fact. In this algorithm, the candidate node with the largest identity is elected as the leader. The complexity of this algorithm is $O(1)$. However, the message complexity is $O(n^2)$, since the number of candidate nodes may be $O(n)$, each of which sends $O(n)$ messages.

In the randomized version of this algorithm, not all candidate nodes send messages to their neighbors. Let r be the number of candidate nodes competing for leadership. Each candidate i tosses a biased coin with probability $p = a/r$ of sending an *elect(i)* message to all nodes where a is a constant. The elect(i) message consists of the ID i. When node j receives an *elect(i)* message it behaves as in the same manner as in the previous algorithm. If j is a candidate and $j > i$, then no response is sent. Otherwise, j sends an *accept* message to i. The node that gets all its *elect* messages acknowledged is declared as the network leader. The algorithm runs in constant time as was in the previous case. In this algorithm the expected number of candidates sending *elect* message is a which is a constant. The expected number of messages is $a.n$, which is *linear.* So we have got a randomized algorithm which is simple and efficient in terms of messages. Since the decision whether to send an *elect* message by a candidate node is taken by tossing a biased coin, it is possible that the algorithm may fail. From the Table 1 it can be seen that our algorithm succeeds in very few number of iterations. More than one iteration is required when the algorithm fails. In that case we have to re-execute the algorithm.

4.3 Analysis

We analyze of the algorithm in this section. The algorithm fails when nobody sends the *elect* message even though there are non-zero number of candidates. Now the probability of failure of the algorithm, that is, the probability of all candidates not sending a message is,

$$x = (1 - p)^r$$

If the algorithm fails then we re-execute the algorithm until we get success. Let s be the number of iterations (re-executions) required to succeed in electing a leader. Now the probability of success after at most s iterations is given by,

$$\begin{aligned} prob &= 1 - x^s \\ &= 1 - (1-p)^{rs} \\ &= 1 - (1 - \frac{a}{r})^{rs} \qquad (4.1) \end{aligned}$$

If we require a confidence interval of $1-\alpha$ for the success of the algorithm, $0 < \alpha < 1$, the expected number of iterations required can be calculated as follows. From (4.1),

$$\begin{aligned} 1-(1-p)^{rs} &= 1-\alpha \\ (1-p)^{rs} &= \alpha \\ rs\ln(1-p) &= \ln\alpha \\ s &= \frac{\ln\alpha}{r\ln(1-p)} \qquad (4.2) \end{aligned}$$

Equation (4.2) gives the expected number of iterations of the algorithm required with a probability of success $1-\alpha$. We can take α to be very small, say 0.001, in practical applications.

Now let us calculate the approximate value of a to be used in the algorithm. Putting $s = 1$ in (4.2), that is, if we want the algorithm to fail with probability of at most α in the first iteration,

$$\begin{aligned} \ln\alpha &= r\ln(1-p) \\ &= \ln(1-p)^r \end{aligned}$$

$$\begin{aligned}\alpha &= (1-p)^r \\ &= (1-\frac{a}{r})^r \\ &\approx e^{-a}\end{aligned}$$

$$a \approx \ln \alpha \tag{4.3}$$

Equation (4.3) holds for large values of r. We can also find out the exact value of a as follows.

$$\begin{aligned}\ln \alpha &= r \ln(1-\frac{a}{r}) \\ \frac{1}{r}\ln \alpha &= \ln(1-\frac{a}{r}) \\ \alpha^{\frac{1}{r}} &= 1-\frac{a}{r} \\ a &= r(1-\alpha^{\frac{1}{r}})\end{aligned} \tag{4.4}$$

From equation (4.1), for 99% confidence $\alpha = 0.01$.

$$1-(1-p)^{rs} = 0.99$$

$$(1-p)^{rs} = 0.01$$

$$rs\log(1-p) = -2$$

$$\begin{aligned}s &= \frac{-2}{r\log(1-p)} \\ &= \frac{-2}{r\log(1-\frac{a}{r})}\end{aligned}$$

The expected number of iterations required for different values of a and r with probability of success 0.99 is given in the table given below. From the table it can be seen that, in practice, the algorithm succeeds in at most five iterations with 0.99 probability of success.

Constant	Number of competitors	Expected number of iterations
a = 1	r = 1	s = 1
	r = 2	s = 4
	r = 4	s = 5
	r = 16	s = 5
	r = 256	s = 5
	r = 65536	s = 5
a = 2	r = 1	s = 1
	r = 2	s = 1
	r = 4	s = 2
	r = 16	s = 3
	r = 256	s = 3
	r = 65536	s = 3
a = 3	r = 1	s = 1
	r = 2	s = 1
	r = 4	s = 1
	r = 16	s = 2
	r = 256	s = 2
	r = 65536	s = 2
a = 4	r = 1	s = 1
	r = 2	s = 1
	r = 4	s = 1
	r = 16	s = 2
	r = 256	s = 2
	r = 65536	s = 2
a = 5	r = 1	s = 1
	r = 2	s = 1
	r = 4	s = 1
	r = 16	s = 1
	r = 256	s = 1
	r = 65536	s = 1

Table 1: Expected number of iterations for different values of a and r

4.4 Scheduling of r - Estimating the number of candidates

We have assumed that each node knows the value of r, the number of competitors, before the execution of the algorithm. Since, in practice this is not known in advance, each node should have an estimate value of r. Let the initial estimate of r be n. Let $f(n)$ be a function of n, the number of nodes. Then the new estimate of r will be $\frac{r_{estimate-old}}{f(n)}$. Throughout the execution of any iteration of the algorithm the estimated value of r is constant. If the algorithm fails then the value of r is changed in the next iteration.

Putting $f(n) = 2$, the scheduling of r will be

$$r_0 = n$$
$$r_1 = \frac{n}{2}$$
$$r_2 = \frac{n}{4}$$

and so on.

Putting $f(n) = \log n$, the scheduling of r will be

$$r_0 = n$$
$$r_1 = \frac{n}{\log n}$$
$$r_2 = \frac{n}{\log^2 n}$$

and so on.

Let us denote the actual value of r by r_{act} and the estimated value of r by r_{est}. The probability of success of the algorithm in the first iteration is given by $Prob = 1 - (1 - \frac{a}{r_{est}})^{r_{act}}$

When the algorithm terminates only one node is elected as the leader. It broadcasts its ID to all nodes telling them that it is the network leader. As seen in Section 4.3, the algorithm will terminate with 99% certainty if $r_{est} \leq r_{act}$ within 5 iterations. In case the algorithm fails, no messages will be sent. When the algorithm terminates $r_{est-prev} > r_{act} \geq r_{est}$ The expected time for first schedule is $O(\log n)$ with linear messages; this can be seen as follows. In case the algorithm fails, no messages will be sent. As soon as $r_{est} > r_{act}$, the probability of success falls down. We expect the algorithm to terminate with reasonable probability if $r_{est} < r_{act}$. So the number of messages sent will be roughly $O(\frac{r_{act}}{r_{est}}n)$; which will be a constant times n for first schedule.

Remark: A randomized algorithm has a *high probability* of success if $Prob = 1 - n^{-c}$, for some positive constant c [14]. Substituting, $1-(1-\frac{a}{r_{est}})^{r_{act}} = 1-n^{-c}$, or $(1-\frac{a}{r_{est}})^{r_{act}} = n^{-c}$, or $e^{-a\frac{r_{act}}{r_{est}}} \approx n^{-c}$, or

$\frac{r_{act}}{r_{est}} \approx \frac{c}{a} \ln n$

Thus, with high probability algorithm uses at most $O(n \log n)$ messages.

4.5 Summary

We have got a simple randomized algorithm which runs in at the most five iterations with a probability of success of 0.99. It was assumed that the number of competitors is known in advance. Later this assumption was relaxed by estimating the value of r. The algorithm is efficient in terms of messages (linear).

Chapter 5

An $O(\log n)$ Time Algorithm for Hypercube Networks

5.1 Introduction

In this chapter we propose an $O(\log n)$ time algorithm for electing a leader in a hypercube network. The algorithm is extended to electing r leaders in the network at the expense of bit complexity. Finally, we see how it can be implemented on a Cube Connected Cycles (CCC) network.

Hypercube networks have been studied widely in both parallel and distributed computations. It is a highly versatile network architecture, which can be used as a general-purpose machine. We model the network by an *undirected graph* $G = (V, E)$. Let n be the number of *nodes* in the network. The IDs of the network are from 0 to $n - 1$. The number of nodes in a hypercube network is an *integral* power of 2. Let $n = 2^m$ or $m = \log n$, all logarithms being to the base 2. For integer x we denote the integers $\{0, 1, ..x - 1\}$ by $< x >$. For integer x we refer to the $(i + 1)^{th}$ *least significant bit (LSB)* in its binary representation $(i = 0, 1, 2, ..)$ as the *"i^{th} bit"* of x. The integer obtained by changing the i^{th} bit if x is denoted by x_i (For instance, if $x = 5$, then $x_0 = 4$, $x_1 = 7$ and $x_2 = 1$). x_i is known as the i^{th}-dimension neighbor of x. The m-cube is defined as (V, E) where $V =< n >$, $n = 2^m$ and $E = \{(w, w_i)|w \in V, i \in< m >\}$. It is also known as the binary m-dimensional hypercube. The graph has n nodes and degree and diameter both equal to m [31]. Two nodes x and y are connected if and only if the binary representations of x and y differ in a single bit.

Alternately, m-cube can be constructed from two $m-1$ cubes by adding one more bit to the IDs at the most significant position (MSB) and completing the additional connections required from the definition. This gives a recursive definition of the hypercube [13]. This idea is used in the design of the algorithms.

We assume that the links are bidirectional. The IDs are distinct. Each node knows its 1^{st}-dimension neighbor, 2^{nd}-dimension neighbor, ...,m^{th}-dimension neighbor. A global clock is connected to all nodes so as to synchronize all computations. We do not consider link failures, messages sent are not lost. At the beginning of the algorithm a subset of nodes decide to compete for leadership. The time and message complexities of the algorithm are the worst case time and messages required in an execution of the algorithm.

5.2 The Algorithm

The algorithm is based on the comparison of IDs. The node having the largest ID among the competitors is elected as the leader. The ID of each node can be can be encoded in $\log n$ bits. A message consists of an ID and a constant number of fields which indicate the type of message. Thus a message consists of $O(\log n)$ bits. The algorithm works in rounds, each round correspond to one clock tick.

The protocol for electing a leader is given next. In the first round, all competing nodes whose first bit (from LSB) is 1 send their IDs to the nodes whose ID differs only in the first bit. Then each node computes the current leader from the message that it gets. If node x gets a message from node y then it computes the current leader as follows. If x is not a candidate for leadership then the current leader is y. If x is a candidate node and $x > y$ then the current leader is x else current leader is y. A non-candidate node which does not get any message will not send any message. In the second round, all nodes whose first bit is 0 and second bit is 1 send the current leader they have computed to nodes whose IDs differ only in the second bit. A non-candidate node will just forward the message that it has got in the earlier round, if any. A candidate node will compare its ID to the ID in the message and sends the current leader. In general, in the i^{th} round all nodes whose first $(i-1)$ bits are 0 and i^{th} bit is 1 send their current leaders to nodes whose ID differ only in the i^{th} bit. This will continue up to $\log n$ rounds, since there are only $\log n$ bits in an ID. After $\log n$ rounds the node whose all bits are 0s will compute its current leader and this will be the network leader. Now the ID of the network leader is broadcasted to all nodes by reversing the protocol, ie. following the communication links in the reverse order. Thus after $2.\log n$ rounds every node knows the ID of the network leader.

We can view the protocol given above in a recursive way. To elect a leader in an m-cube, elect leaders in each of the two $m-1$-cubes. Then with a single message the network leader can be determined (from the protocol). We do the same procedure for electing leaders in the $(m-1)$-cubes. A leader is elected in a 1-cube by simply sending a message from one node to the other (trivial step). Let $T(n)$ be the time complexity of the algorithm for electing a leader in a cube containing n nodes. Similarly let $M(n)$ be the maximum number of messages required. Now we have the following recurrence relations.

$$\begin{aligned} T(n) &= T(\frac{n}{2}) + O(1) \\ &= O(\log n) \\ M(n) &= 2.M(\frac{n}{2}) + 1 \\ &= O(n) \end{aligned}$$

Thus the time complexity of our algorithm is $O(\log n)$ and the message complexity is $O(n)$.

5.3 Electing r Leaders

We generalize the problem of electing a single leader to electing r leaders in a network, $r \leq n$. There are situations where it is necessary to have more than one leader for doing some useful operation. An example is the case when we have to elect some nodes where the replicated copies of a database are to be stored.

In electing r leaders, we use the same protocol except that a message may contain $O(r \log n)$ bits. The message consists of a field left for r leaders. A node simply inserts its ID to a message got from neighboring node. If the field is already full, then a decision whether to insert its ID is made by comparing its ID to the list of current leaders in the list. At the end of $\log n$ rounds the r leaders are calculated and broadcasted to every node, again requiring $O(n)$ messages.

The time complexity and the message complexity remains the same, but the bit complexity is $O(nr \log n)$ where in the previous algorithm (election of a single leader) it was $n \log n$.

$$T(n) = O(\log n)$$

$$M(n) = O(n)$$

5.4 Summary

In this chapter we have proposed an algorithm for electing a leader in a hypercube network with running time $O(\log n)$ and and with *linear* number of messages. The algorithm was extended to electing r leaders at the expense of bit complexity (due the increased message size).

Appendix: Implementing the Algorithm on Cube Connected Cycles (CCC)

Cube connected cycles architecture was proposed in [20] as a practical network for parallel computation and as a general purpose parallel processor. The cube, which has been studied extensively in relation to parallel computation, is not readily usable in VLSI design, since each of the 2^m processors in the system is connected to m other processors. The Cube Connected Cycles (CCC) network is a feasible substitute for the cube connected network [20]. It has all the desirable features and a VLSI layout which is not only more compact but more regular. It is demonstrated that for a wide class of problems the CCC is optimal with respect to the $area \times (time)^2$ measure of complexity in the VLSI model. The operation of the CCC is based on the combinations of pipelining and parallelism, which leads to the following results.

- The number of connections per processor is reduced to three.
- Processing time is not significantly increased with respect to that achievable on the cube connected network.

- The overall structure compiles with the basic requirements of VLSI technology.
- Programs for the individual modules are obtained in a systematic way from a standard description of the global algorithms.
- Finally, without resorting to any drastic departure from classical ALGOL like languages, fully accurate and hopefully, easily understandable descriptions of parallel programs can be provided.

CCC is a network of processors, with each processor having at most three neighbors. An easy way of understanding the network architecture is given below. Suppose we have an s-cube which has 2^s nodes, each node having a degree s. Now we replace each node by a cycle containing h nodes where $h \geq s$. These cycles are connected as s-dimensional cube. Now total number of nodes, $n = h.2^s$.

ASCEND and DESCEND are classes of algorithms that run efficiently on parallel machines. Assume that the input data $t_0, t_1, t_2, ..., t_{n-1}$ are stored respectively, in storage locations $T[0], T[1], T[2], ..., T[n-1]$ and that $n = 2^k$, ie. the number of inputs is a power of 2. An algorithm is in the DESCEND class if it performs a sequence of basic operations on pairs of data that are successively $2^{k-1}, 2^{k-2}, ..., 2^0 = 1$ locations apart. Each basic operation $OPER(m, j; U, V)$ modifies the two data items present in storage locations U and V; the computations performed affects only the contents of U and V and may depend on parameters m and j, which are integers $0 \leq m \leq n, 0 \leq j < k$.

Algorithms in the DESCEND class are then specified as in Figure 1.
Let $bit_j(m)$ be the the coefficient of 2^j in the binary representation of $m = \sum_{j \geq 0} bit_j(m) 2^j$. The language construct *foreach* indicates that all instructions are performed simultaneously. On machines where such parallelism can be realized, DESCEND algorithms run in $k = \log_2(n)$ elementary steps.

```
proc DESCEND
      for $j \leftarrow k-1$ step $-1$ until $j=0$
      do foreach $m : 0 \leq m < n$
         pardo if $bit_j(m) = 0$ then $OPER(m, j; T[m], T[m+2^j])$
               fi
         odpar
      od
corp DESCEND
```

Figure 1: DESCEND class of algorithm

ASCEND is the dual class of DESCEND. For ASCEND the control of the algorithm is changed to

for $j \leftarrow 0$ step 1 until $j = k-1$

ie. $OPER$ is performed on data that are successively $l = 2^0, 2^1, ..., 2^{k-1}$ locations apart.

To clarify the duality between ASCEND and DESCEND, consider the binary representation of $m = \sum_{0 \leq i < k} bit_i(m).2^i$ and define $m' = \sum_{0 \leq i < k} bit_i(m).2^{k-i-1}$, the integer whose binary representation is the reversal of that of m. Once k is fixed, the function $m \rightarrow m'$ is an involutory permutation of $0, 1, ..., 2^k - 1$ known as the bit reversal permutation (BRP). For example, if $(k = 3)$, then the BRP of $(0,1,2,3,4,5,6,7)$ is $(0,4,2,6,1,5,3,7)$. By first applying BRP to its inputs, an ASCEND algorithm can be transformed into a dual DESCENT algorithm whose basic operation $OPER'$ is related to the original $OPER$ by

$$OPER'(m, j; U, V) = OPER(m', k-1-j; U, V)$$

These algorithms run in $O(\log n)$ steps.

In order to efficiently implement algorithms in the DESCENT or ASCEND classes, the most natural interconnection of nodes is the m-dimensional binary cube (m-cube), where each of the 2^m processors is numbered from 0 to $2^m - 1$ and is connected to each of m processors whose binary numbering differs in exactly one binary position. Although an ASCEND or DESCEND algorithm can be implemented on such a machine in $\log_2 n$ parallel steps, this is not feasible mainly because the number $m = \log_2 n$ of connections for each processor is too large. The CCC architecture was suggested as a practical network for easier VLSI implementations. CCC as defined belongs to the hypercube family.

The algorithm we have proposed for electing a leader in a hypercube network belongs to the ASCEND class. CCC can successfully emulate the m-cube in executing ASCEND or DESCEND algorithms by combining the principles of parallelism and pipelining with no significant degradation of performance but with a more compact structure. The interested reader is asked to refer to [20] for the details of the emulation algorithm.

Chapter 6

Conclusions

In this thesis we have proposed new leader election algorithms in distributed systems. Algorithms were designed for ring, complete, hypercube and CCC networks. All algorithms were designed to work in a synchronous model. The idea of electing a single leader in a network was extended to the election of r leaders in a network. Algorithms for electing r leaders were designed for ring and hypercube networks, whereas it was found that in complete networks both problems are equivalent in terms of complexities. Proposed algorithms for electing r leaders run with same time and message complexities, but at the expense of bit complexity. The general algorithm designed for electing a leader in complete networks by using the best features of the two existing algorithms was found to be optimal, with the time-message trade-off. The idea of randomization was found to be useful in getting better and simple algorithms for complete networks. The $O(\log n)$ time recursive algorithm proposed for hypercube networks was found to be in the ASCEND class of algorithms, so that it can easily be implemented on other networks in hypercube family - for example, CCC. All synchronous algorithms proposed in this thesis can be implemented on practical asynchronous networks by using synchronizers (see Appendix A).

We next mention some open problems and give directions for further work in this area. It was found that in some networks, the problem of electing a single leader and electing r leaders were found to be equivalent in terms of complexities. It is interesting to investigate further in what topologies these problems are equivalent. It seems intuitively that these problems are equivalent in networks where routing will not take much time and messages. Another open problem is that of electing r leaders in general graphs, that is, in networks where the processors are connected in an arbitrary fashion. All algorithms in this thesis were designed for synchronous model. It would be interesting to extend these algorithms to work on asynchronous networks without using synchronizers. Since the idea of randomization was found to be useful in getting better and simple algorithms, this area can be explored further. Another open problem is that of electing r leaders in anonymous networks. In our algorithms we assumed that the processors and the links do not fail. It would be interesting to make our algorithms fault-tolerant.

Appendix A

Implementing Synchronous Algorithms in Asynchronous Networks

A.1 Synchronizers

The implementer of any system is faced with the situation that the goal system must satisfy stronger requirements than the underlying hardware. Two models of computation have been used for the development of distributed algorithms: the *synchronous model* and the *asynchronous model*. In the synchronous model the execution of an algorithm operates in discrete steps called *rounds*. The actions of a process in round $(i+1)$ depend on the state of the process after round i and the messages sent to it in round i. It is therefore necessary that all messages that are sent to some process in round i are received before the process starts its computation of round $(i+1)$. We can think of the system as if there were a global clock, giving pulses at regular intervals. Computation takes place at clock pulses, and a message that is sent at one pulse is guaranteed to be received before the next pulse. In the asynchronous model it is assumed that there are no clocks and message delivery time is not bounded a priori.

All algorithms in this thesis were designed for the synchronous model. The synchronous model is stronger than the asynchronous model. Consequently, distributed algorithms for synchronous networks are more efficient and easier to design than algorithms for asynchronous networks. Synchronous models are of much theoretical importance. Practical networks are asynchronous. Therefore simulation algorithms have been designed to simulate synchronous computations on asynchronous networks. These simulation algorithms are called *synchronizers.* Using synchronizers a synchronous algorithm can be turned into an asynchronous algorithm at the expense of additional message and time complexities, so that the lack of a global time basis and of bounds on delays for message delivery can be dealt with, and the resulting algorithm can be guaranteed to function as in the synchronous model. Synchronizers ensure that exactly one message is sent over each link of the network in every round. If the simulated algorithm sends more messages over some link in some round, these messages must be packed into one larger logical message. If the simulated algorithm sends no messages over some link in some round, a special "empty message" must be sent [30]. As a result of this policy, every process must receive exactly one message from every neighbor after every round. The next round is simulated when the message of the current round has been received from all neighbors. The addition of empty messages makes the synchronizer inefficient for computations that are "sparse" in time. The message complexity of the simulated algorithm equals the time complexity multiplied by the number of edges in the network. The synchronizer described above is optimum in terms of time but it can increase the message complexity of the algorithm to exponential.

Until now our assumption was that there is no upper bound on the message delivery time. If we assume that there is an upper bound on the maximum delay in message delivery this leads to a new network model called *Asynchronous Bounded Delay networks (ABD networks)* /citeref14. This model is weaker than the synchronous model, but stronger than the asynchronous model. It is assumed that processes have local clocks, but they are not synchronized. That is, they need not show the same value at one instant. Furthermore a fixed bound on message delivery time is assumed. We can choose our unit of time equal to this bound and assume henceforth that message delay is bounded by 1.

Formally, if σ is the global time of the sending of a message, and τ is the global time of its receipt, then

$$\sigma \leq \tau < \sigma + 1$$

In ABD networks a synchronizer can work without any empty messages. An initial exchange of START message is required to make every process start its local clock at approximately the same time. The following two requirements must be satisfied.

- If a process q sends a message to its neighbor p in some round i, this message must be received before p simulates round $(i+1)$.
- If a process p receives a message it must be possible for p to determine to what round this message belongs.

First requirement must be satisfied because p's actions in round $(i+1)$ depend on q's message. Failure to meet second requirement may lead to incorrect simulation. If the two requirements are satisfied the synchronous computation is simulated correctly. *Round time* of a synchronizer is the time it takes to simulate one round of the synchronous algorithm. This is used to compare the speed of synchronizers. When the simple synchronizer, described earlier, is used on an ABD network, it realizes a round time of 1. This simple synchronizer is time-optimal, but it uses a lot of messages.

Many synchronizers have been proposed in the literature. (see for example, [5, 30]). Using synchronizers we can implement the synchronous algorithms that we have proposed in this thesis, on practical asynchronous networks.

ARY
A 125387

References

[1] K. Abrahamson, A. Adler, R.Gelbart, L. Higham and D. Kirkpatrick, "The Bit Complexity of Randomized Leader Election on a Ring", *SIAM J. Computing*, Vol 18, Feb 1989, 12-29

[2] Y. Afek and E. Gafni, "Time and message bounds for election in synchronous and asynchronous complete networks", *SIAM J. Computing*, 20(2), 1991, 376-394.

[3] Y. Afek, "Elections in Anonymous Networks", *Information and Computation*, 113, 312-330, (1994)

[4] Y. Afek and G. Stupp, "Optimal Time-Space Tradeoff for Shared Memory Leader Election", *J. of Algorithms*, 25, 95-117, (1997)

[5] V.C. Barbosa, *An Introduction to Distributed Algorithms*, The MIT Press, 1995.

[6] L. Bodlaender and G. Tel, "Bit-Optimal Election in Synchronous Rings", *Info. Processing Letters*, 36(1990), 53-56

[7] J. Brunekreef, J. Katoen, R. Koymans, S. Mauw, "Design and analysis of dynamic leader election protocols in broadcast networks", *Distributed Computing*, (1996) 9: 157-171

[8] E. Chang and R. Roberts, "An improved algorithm for decentralized extrema finding in circular arrangements of processes", *Comm. ACM*, 22(1979), 281-283

[9] G. Fredrickson and N. Lynch., "Electing a leader in a synchronous ring", *J. Assoc. Comput. Mach.*, 34, 1 (1987), 48-59. *SIAM J. Computing*, 20(2), 1991, 376-394.

[10] P. Francis and S. Saxena, "Efficient Algorithm for Electing r Leaders in a Synchronous Ring", *Proceedings of the Seventh National Seminar on Theoretical Computer Science* , June 1997, C74-C77.

[11] H. Garcia-Molina., "Elections in a Distributed Computing System", *IEEE Trans. on Computers*, Vol C-31, Jan 1982, 48-59.

[12] R.G. Gallager, P.A. Humblet, P.M. Spira, "A distributed algorithm for minimum-weight spanning trees", *ACM Trans. on Prog. and Lang. and Systems*, 5(1983), 67-77

[13] F. Harary, *Graph Theory*, Addison-Wesley, 1969.

[14] J. Jaja, *An Introduction to Parallel Algorithms*, Addison-Wesley, 1992

[15] E. Korach, S. Kutten, S. Moran, "A modular technique for the design of efficient distributed leader finding algorithms", *ACM Trans. on Prog. Lang and Systems*, 12(1990), 84-101

[16] T.Z. Kalamboukis and S.L. Mantzaris, "Towards optimal election on chordal rings", *Info. Processing Letters*, 38(1991), 265-270

[17] G. LeLann, "Distributed systems - towards a formal approch", *Inf. Processing*, 1977, 155-160

[18] R. Motwani and P. Raghavan, *Randomized Algorithms*, Cambridge University Press, 1995.

[19] A. Marchetti-Spaccamela, "New Protocols for the Election of a Leader in a Ring", *Theoretical Computer Science*, 54(1987), 53-64

[20] F.P. Preperata and J. Vuillemin, "The Cube-Connected Cycles: A Versatile Network for Parallel Computation", *Comm. of ACM*, May 1981, Vol 24, No.5, 300-309.

[21] G.L. Peterson, "An $O(n \log n)$ unidirectional algorithm for the circular extrema problem", *ACM Trans. on Prog. Lang. and Systems*, 4(1982), 758-762

[22] S. Singh and J. Kurose., "Electing good leaders", *J. Parallel and Distributed Computing*, 21, (1994), 184-201.

[23] G. Singh, "Leader Election in the Presence of Link Failures", *IEEE Trans. on Parallel and Distributed Systems*, Vol 7, No.3, March 1996.

[24] G. Singh, "Leader Election in Complete Networks", *ACM Symp. Principles of Distributed Computing*, 1992, 179-190.

[25] G. Singh, "Leader Election in Complete Networks", *SIAM J. Computing*, Vol 26, No.3, June 1997, 772-785.

[26] G. Singh, "Leader Election in the Presence of Link Failures", *IEEE Trans. on Parallel and Distributed Systems*, Vol 7, No.3, March 1996

[27] G. Singh, "Efficient leader election using sense of direction", *Distributed Computing*, (1997) 10: 159-165

[28] G. Singh, "Real-time leader election", *Info. Processing Letters*, 49(1994), 57-61

[29] H. M. Sayeed, M. Abu-Amara, H. Abu-Amara, "Optimal asynchronous agreement and leader election algorithm for complete networks with Byzantine faulty links", *Distributed Computing*, (1995) 9: 147-156

[30] G. Tel, *Topics in Distributed Algorithms*, Cambridge International Series:1, Cambridge University Press, 1991.

[31] L. G. Valiant, "General Purpose Parallel Architectures", *The Handbook of Theoretical Computer Science, Vol 1: Algorithms and Complexity*, Edited by J. van Leeuwen, MIT Press, Cambridge, Massachusetts, 1990, 943-971.